LLNL-CONF-413969

Frontiers of Performance Analysis on Leadership-Class Systems

R. J. Fowler, L. Adhianto, B. R. de Supinski, M. Fagan, T. Gamblin, M. Krentel, J. Mellor-Crummey, M. Schulz, N. Tallent

June 16, 2009

SciDAC Conference
San Diego, CA, United States
June 16, 2009 through June 18, 2009

Frontiers of performance analysis on leadership-class systems

R Fowler[1], L Adhianto[2], B de Supinski[3], M Fagan[2], T Gamblin[3], M Krentel[2], J Mellor-Crummey[2], M Schulz[3], and N Tallent[2]

[1] Renaissance Computing Institute, UNC, Chapel Hill, North Carolina
[2] Rice University, Houston, Texas
[3] Lawrence Livermore National Laboratory

E-mail: `rjf@renci.org, laksono@cs.rice.edu, bronis@llnl.gov, mfagan@cs.rice.edu, tgamblin@llnl.gov, krentel@cs.rice.edu, johnmc@cs.rice.edu, schulzm@llnl.gov, tallent@cs.rice.edu`

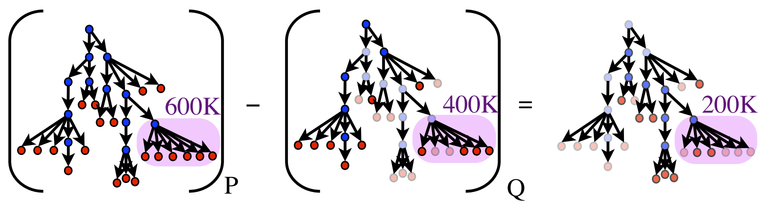
Abstract. The number of cores in high-end systems for scientific computing are employingis increasing rapidly. As a result, there is an pressing need for tools that can measure, model, and diagnose performance problems in highly-parallel runs. We describe two tools that employ complementary approaches for analysis at scale and we illustrate their use on DOE leadership-class systems.

1. Extreme Scaling and its Performance Challenges

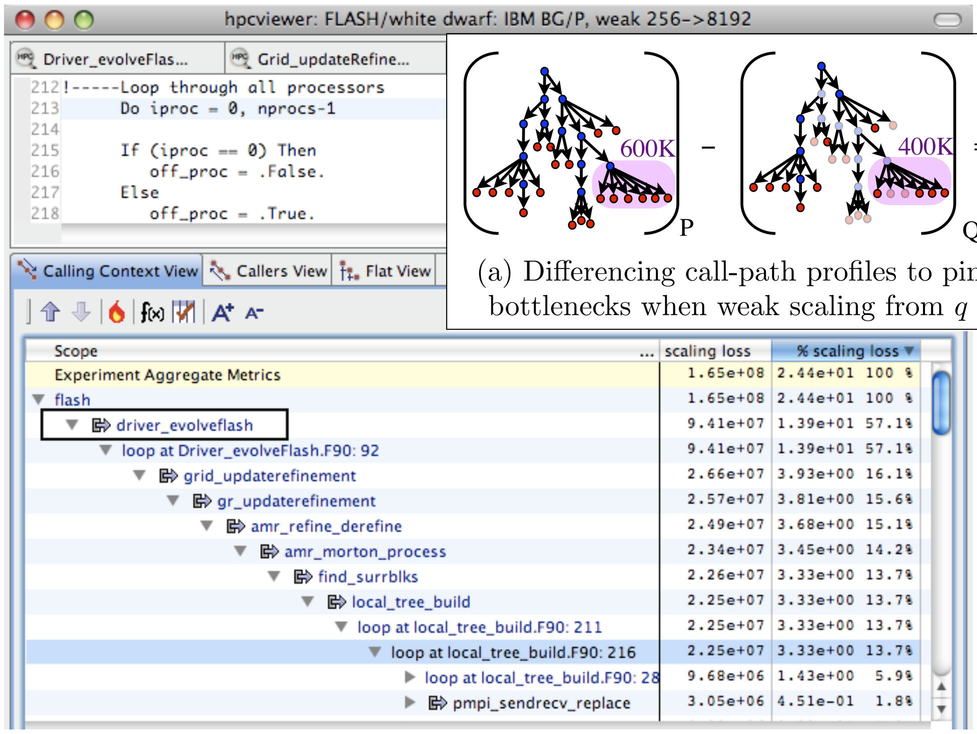
In recent years, the emergence of multi-core computer chips has been a factor in the dramatic increase in the total number of cores, or computational elements (CEs), deployed in top-end systems. In the November, 2008 Top500 list [1], 12 systems had over 32K cores, and one had over 200K. Systems are being designed to scale to over one million. Data parallelism using an SPMD programming style has been remarkably successful at scaling applications up to thousands of cores, but computational science researchers who aspire to use very large systems efficiently must face the harsh reality of Amdahl's law: as the count of CEs goes up, every part of an application must exhibit appropriate scaling behavior on every CE or else there will be a significant loss in performance. Thus, it is vital to characterize the scaling properties of all phases of the computation. The characterization must cover all of the CEs over the entire duration of a run. Furthermore, it is necessary for tools to present specific details about performance problems that must be addressed to achieve the required efficiency at scale.

There does not yet exist a practical approach that can simultaneously achieve both the needed coverage and detail with a single set of measurements obtained with a single tool. To do so would require recording, communicating, and storing a massive quantity of data that grows with the sizes of system and application as well as with the duration of a problem run. Even if it were possible to collect the data, analysis of the data would be daunting and expensive.

We describe practical tools that address tractable and complementary aspects of the problem, together covering a large part of the problem space. These tools and methods were developed with DOE SciDAC support. We illustrate the use of these tools through examples that apply them to scientific codes that run on the DOE Office of Science leadership-class systems.



(a) Differencing call-path profiles to pinpoint scalability bottlenecks when weak scaling from q to p processors.



```
!-----Loop through all processors
      Do iproc = 0, nprocs-1

        If (iproc == 0) Then
          off_proc = .False.
        Else
          off_proc = .True.
```

Scope	scaling loss	% scaling loss
Experiment Aggregate Metrics	1.65e+08	2.44e+01 100 %
flash	1.65e+08	2.44e+01 100 %
driver_evolveflash	9.41e+07	1.39e+01 57.1%
loop at Driver_evolveFlash.F90: 92	9.41e+07	1.39e+01 57.1%
grid_updaterefinement	2.66e+07	3.93e+00 16.1%
gr_updaterefinement	2.57e+07	3.81e+00 15.6%
amr_refine_derefine	2.49e+07	3.68e+00 15.1%
amr_morton_process	2.34e+07	3.45e+00 14.2%
find_surrblks	2.26e+07	3.33e+00 13.7%
local_tree_build	2.25e+07	3.33e+00 13.7%
loop at local_tree_build.F90: 211	2.25e+07	3.33e+00 13.7%
loop at local_tree_build.F90: 216	2.25e+07	3.33e+00 13.7%
loop at local_tree_build.F90: 28	9.68e+06	1.43e+00 5.9%
pmpi_sendrecv_replace	3.05e+06	4.51e-01 1.8%

(b) Using HPCTOOLKIT to analyze scaling losses (microseconds) for FLASH when running at 256 and 8192 cores on an IBM BG/P.

Figure 1. Pinpointing scalability bottlenecks using differential analysis of call path profiles.

2. Measuring and Characterizing Scalability.

2.1. Differential Call-Path Profiling using HPCTOOLKIT

If an execution of a data-parallel program is sufficiently balanced and symmetric, the behavior of the program as a whole can be characterized by examining the scaling performance of any one of the CEs. If behavior is sufficiently stable over time, then an entire run can be characterized by analyzing profiles that are integrated over any appropriate time interval.

Differential profiling [2] is a strategy for analyzing multiple executions of a program by combining their execution profiles mathematically. A comparison of profiles from two executions yields information about where and how much the costs in the executions differ. While a flat profile difference may indicate that more time is spent in some procedure X in one execution relative to another, if X is called from multiple places, a flat profile difference lacks information about how the extra cost in X was incurred on behalf of different calls to X.

To understand an execution's costs in context, Rice University's HPCTOOLKIT uses call-path profiling [3]. HPCTOOLKIT's call-path profiler uses call-stack unwinding in conjunction with sampling to attribute execution costs to full calling contexts [4]. This capability is particularly important for procedures such as `MPI_Wait`, whose costs are very dependent on calling context.

To pinpoint and quantify scalability bottlenecks for parallel programs and to attribute them to calling contexts, we compare call path profiles from a pair of executions using different levels of parallelism [5]. Consider two parallel executions of an application, one executed on q processors and the second executed on p processors, where $p > q$. In a weak scaling scenario, each processor in both executions computes on an identical amount of local data. If the application exhibits perfect weak scaling, then the total cost (*e.g.*, execution time) should be the same in both executions. If every part of the application scales uniformly, then this equality should hold in *each scope* of the application. Figure 1(a) pictorially shows the analysis of weak scaling by comparing two representative calling context trees (CCTs)[1]—one tree from a process in each execution. For instance, the difference in cost incurred in the subtree highlighted in magenta in the CCTs represents parallel overhead when scaling from q to p processors. This difference in cost for the subtree can be converted into percent scalability loss by dividing by the inclusive cost of the root node in the CCT for p processors and multiplying by 100. Computing inclusive scalability losses for each node in a CCT enables one to locate scalability bottlenecks easily in a top-down fashion by tracing patterns of losses down paths from the root.

2.2. Scalable Analysis of Dynamic Load-Balance using Libra

Characterizing and tuning perfectly balanced SPMD codes, however, is only part of the problem. Many large-scale codes use irregular meshes, so partitioning of problems is not perfect. Methods such as adaptive mesh refinement (AMR) have behavior that evolves over time and the evolution induces load imbalances over time. Such behavior can be highly dependent on input data and the form of imbalances may change from problem to problem. The run-time layout of the application on the hardware can also introduce imbalances due to imperfect mappings of application topology to the system, causing differences in communication and I/O costs among the population of CEs. Finally, balance and efficiency can be impacted by run-time conditions such as contention for communication and I/O between jobs that happened to be scheduled together, or by the overhead of using error-correcting mechanisms to tolerate memory or interconnect (*e.g.*, Infiniband) problems. The challenge of dealing with these problems requires methods that can track economically the time evolution of scalable computations while still retaining enough detail to identify and diagnose anomalous behavior.

In prior work, we developed the Libra [6, 7] load-balance measurement tool. In the presence of adaptive and dynamic behavior, tools must be capable of characterizing the evolution of imbalances and of localizing the sources of imbalance to semantically meaningful parts of the application. We therefore use a model in which performance metrics are presented as *rates* normalized to a hierarchical decomposition of the application's behavior. *Progress* steps correspond to program phases that represent application-defined progress towards completing the computation. A time step in a simulation is a typical progress step. A hierarchy of *effort* steps can be nested within each of the progress steps. Effort steps represent the phases of work done to make a unit of progress. Effort steps in large parallel programs correspond to communication actions (especially collective operations), each iteration of an implicit solver, local physics computations, I/O operations, etc.

Libra uses the PMPI interface to attach measurement operations to synchronous MPI calls. These are used to denote the end points of effort regions automatically. Currently, the user must mark each progress region by manualling adding a call to a Libra library routine. We are investigating techniques to automate this process.

For each effort region, Libra generates a two-dimensional profile representing effort in that region across processes and over time. The data from these regions can be copious, so we use aggressive, parallel wavelet compression to aggregate and communicate these data to a central location. Our compression algorithm is lossy, but it preserved significant features even at very low bit rates. Figure 2 shows Libra's compression pipeline in detail.

This strategy allows us to achieve between two and three orders of magnitude of compression, typically, while still preserving outliers, load distribution, and evolutionary behavior. Our compression algorithm also exhibits good scaling behavior. Each region is compressed independently, thus permitting the work of compression to be balanced evenly across all processes

[1] A CCT is a weighted tree in which each calling context is represented by a path from the root to a node and a node's weight represents the cost attributed to its associated context.

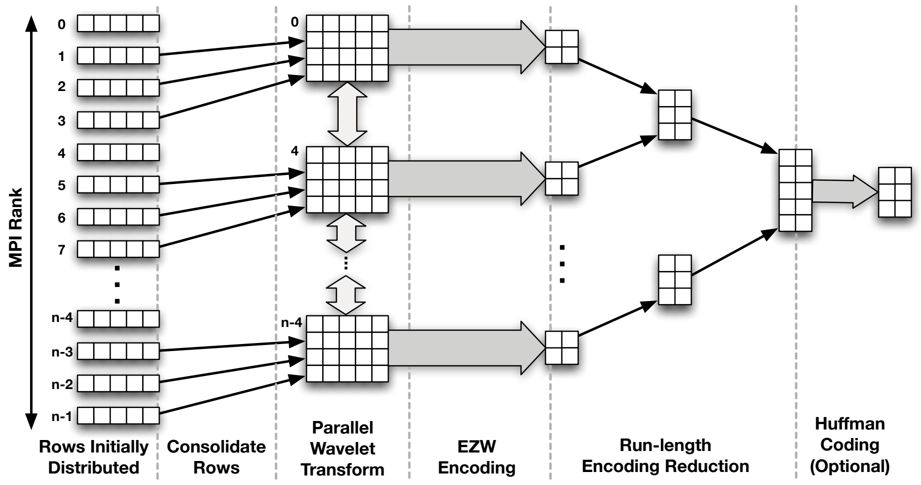

MPI Rank
0
1
2
3
4
5
6
7
n-4
n-3
n-2
n-1
0
4
n-4
Rows Initially Distributed
Consolidate Rows
Parallel Wavelet Transform
EZW Encoding
Run-length Encoding Reduction
Huffman Coding (Optional)

Figure 2. Libra Compression Pipeline

in the application, The time required for compression does not increase significantly as system size grows.

3. Case Studies

We present brief examples illustrating the application of these tools to real problems.

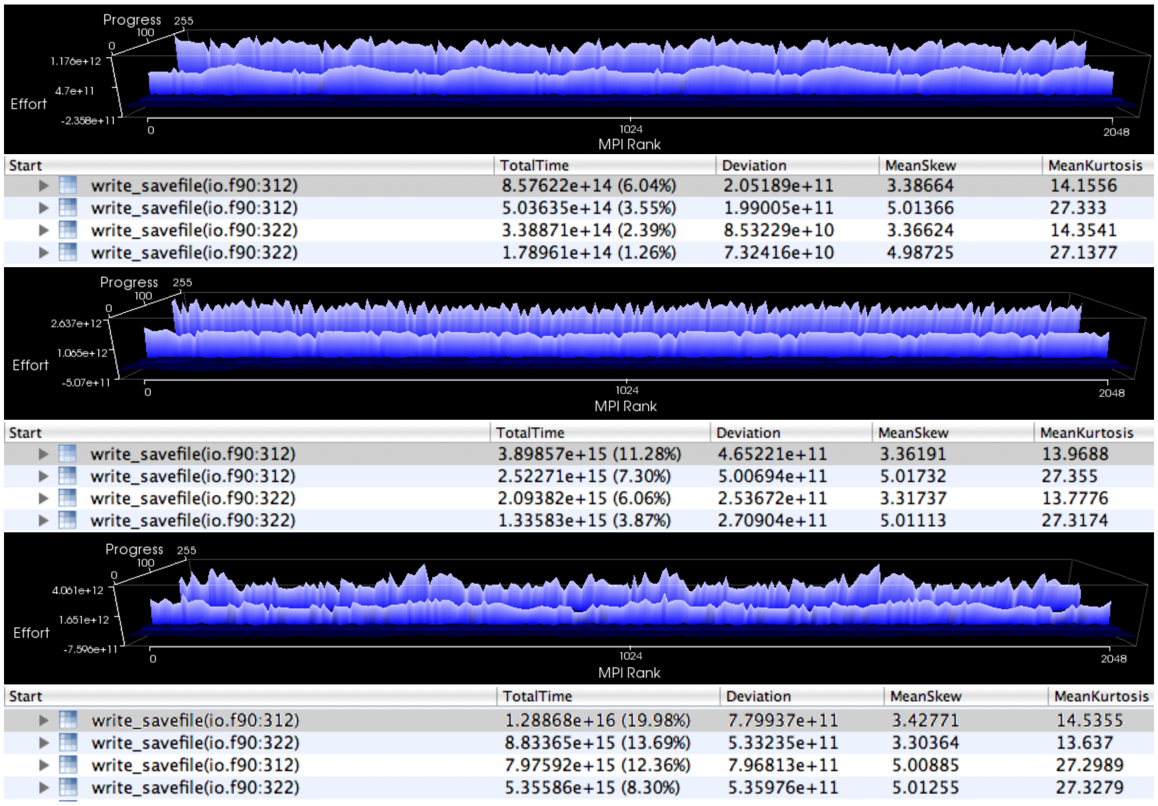
3.1. Pinpointing Scalability Bottlenecks using HPCToolkit

To show the power of HPCToolkit's differential call-path profiling analysis, we used it to pinpoint scaling bottlenecks in FLASH [8], a code that uses adaptive mesh refinement to model astrophysical thermonuclear flashes. We performed a weak scaling study of a white dwarf explosion by executing 256-core and 8192-core simulations on an IBM BlueGene/P located at the Argonne Leadership Computing Facility. For the 8192-core execution, both the input and the number of cores are 32x larger than the 256-core execution. With perfect scaling, we would expect identical run times and call-path profiles for both configurations.

Figure 1(b) shows a portion of the residual calling-context tree, annotated with two metrics: "scaling loss" and "% scaling loss". The former quantifies the scaling loss (in microseconds) while the latter expresses that loss as a percent of total execution time (shown in scientific notation). The top-most entry in each metric column gives the aggregate metric value for the whole execution. A percentage to the right of a metric value indicates the magnitude of that particular value relative to the aggregate. Thus, for this execution, there was a scaling loss of 1.65×10^8 μs, accounting for 24.4% of the execution.[2] By sorting calling contexts according to metric values, we immediately see that the evolution phase (`Driver_evolveFlash`) of the execution (highlighted with a square) has a scaling loss that accounts for 13.9% of the total execution time on 8192 cores, which represents 57.1% of the scaling losses in the execution.

To pinpoint the scalability bottleneck in FLASH's simulation, we use the "hot path" button to expand automatically the unambiguous portion of the hot call path according to this metric. Figure 1(b) shows this result. HPCToolkit identifies a loop (beginning at line 213), within its full dynamic calling context – a unique capability – that is responsible for 13.7% of the scaling losses. This loop uses a ring-based all-to-all communication pattern known as a *digital orrery* to update a processor's knowledge about neighboring blocks of the adaptive mesh. Although FLASH only

[2] A scaling loss of 24.4% means that FLASH is executing at 75.6% parallel efficiency on 8192 cores relative to its performance on 256 cores.



Start	TotalTime	Deviation	MeanSkew	MeanKurtosis
write_savefile(io.f90:312)	8.57622e+14 (6.04%)	2.05189e+11	3.38664	14.1556
write_savefile(io.f90:312)	5.03635e+14 (3.55%)	1.99005e+11	5.01366	27.333
write_savefile(io.f90:322)	3.38871e+14 (2.39%)	8.53229e+10	3.36624	14.3541
write_savefile(io.f90:322)	1.78961e+14 (1.26%)	7.32416e+10	4.98725	27.1377

Start	TotalTime	Deviation	MeanSkew	MeanKurtosis
write_savefile(io.f90:312)	3.89857e+15 (11.28%)	4.65221e+11	3.36191	13.9688
write_savefile(io.f90:312)	2.52271e+15 (7.30%)	5.00694e+11	5.01732	27.355
write_savefile(io.f90:322)	2.09382e+15 (6.06%)	2.53672e+11	3.31737	13.7776
write_savefile(io.f90:322)	1.33583e+15 (3.87%)	2.70904e+11	5.01113	27.3174

Start	TotalTime	Deviation	MeanSkew	MeanKurtosis
write_savefile(io.f90:312)	1.28868e+16 (19.98%)	7.79937e+11	3.42771	14.5355
write_savefile(io.f90:322)	8.83365e+15 (13.69%)	5.33235e+11	3.30364	13.637
write_savefile(io.f90:312)	7.97592e+15 (12.36%)	7.96813e+11	5.00885	27.2989
write_savefile(io.f90:322)	5.35586e+15 (8.30%)	5.35976e+11	5.01255	27.3279

Figure 3. Load balance plots for checkpointing operations in S3D runs at 4096 (top), 8192 (middle), and 16384 (bottom) cores on an IBM BG/P.

uses the orrery pattern to set up subsequent communication to fill guard cells, looping over all processes is inherently unscalable. Other scalability bottlenecks can be identified readily by repeating the "hot path" analysis for other subtrees in the computation where scaling losses are present. The power of HPCToolkit's scalability analysis is apparent from the fact that it immediately pinpoints and quantifies the scaling loss of a key loop deep inside the application's layers of abstractions.

3.2. Examining Load Balance Using Libra

We used Libra to measure the load-balance behavior of S3D, a Direct Numerical Simulation (DNS) of turbulent combustion. Fine-grain calculations to simulate micro-turbulence with S3D can be used to generate coarser models for engineering-level simulation codes.

The computational regions of S3D exhibit almost perfect weak scaling [9]. We applied Libra to 200-time step runs of S3D on a Blue Gene/P system (Intrepid) at Argonne National Laboratory's Leadership Computing facility. For each time step, we examined the time spent in `MPI_Wait` and `MPI_Barrier`, uniquely identifying each call-site by its full call-path labeled with file names and line numbers for each frame. We confirmed that that `MPI_Wait` and `MPI_Barrier` communication routines dominated the runtime of S3D at large scales, and that most of this time is attributed to two call-sites in S3D's `write_savefile` checkpoint routine. Since `write_savefile` is called from two places in the code, four call-paths dominated execution time.

Figure 3 shows load balance profiles reported by Libra for MPI operations used in checkpointing for 4096, 8192, and 16384-process runs (top to bottom) of S3D on Intrepid. The tables below the profiles present summary statistics for each call-site. In the plots the vertical axis represents time, the depth axis represents time steps (progress), and the horizontal axis represents processes by MPI rank. In each image there are two "ridgelines" parallel to the page corresponding to two of the periodic checkpoint operations. As we increased the size of the runs, the plots and summaries reveal a large increase in total time spent in checkpointing. I/O imbalance (inter-process variation) also increases with the number of nodes. This version of S3D writes a separate checkpoint file per process. Some processes quickly write their data to disk, while others encounter contention delays in writing theirs, resulting in the sawtooth load patterns seen in the figures. We are currently investigating and measuring runs of S3D with alternative I/O strategies designed to reduce and balance contention during the checkpoint phase.

4. Conclusions and Future Directions

The two approaches that we describe are complementary. We are currently investigating the question of coordinating their usage. For example, a unified approach would have Libra dynamically enable HPCToolkit profiling to track problems in detail.

An emerging challenge is that explicit shared-memory parallelism on multi-core chips is contributing a substantial multiplicative factor to the overall performance of large systems. This adds a new dimension to the complexity of performance analysis.

Acknowledgments

This work was supported by the DOE SciDAC program under cooperative agreement numbers DE-FC02-06ER25764, DE-FC02-06ER41445, DE-FC02-07ER25800, and DE-FC02-06ER25762. LLNL release number LLNL-CONF-413969.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the DOE Office of Science under contract DE-AC02-06CH11357. Prepared by LLNL under Contract DE-AC52-07NA27344.

References

[1] Meuer H, Strohmaier E, Dongarra J and Horst S Top500 supercomputer sites (http://www.top500.org)

[2] McKenney P E 1998 *Software: Practice and Experience* **29** 219–234

[3] Hall R J 1992 Call path profiling *Proc. of the* 14^{th} *Intl. Conf. on Software Engineering* (NY, NY, USA: ACM) pp 296–306

[4] Tallent N, Mellor-Crummey J and Fagan M 2009 Binary analysis for measurement and attribution of program performance *Proc. of the ACM SIGPLAN Symp. on Program Language Design and Implementation* (NY, NY, USA: ACM)

[5] Coarfa C, Mellor-Crummey J, Froyd N and Dotsenko Y 2007 Scalability analysis of SPMD codes using expectations *Proc. of the 21st Annual Intl. Conf. on Supercomputing* (NY, NY, USA: ACM) pp 13–22

[6] Gamblin T, de Supinski B R, Schulz M, Fowler R J and Reed D A 2007 Scalable load-balance measurement for SPMD codes *In Submission*

[7] Gamblin T 2009 *Scalable Performance Measurement and Analysis* Ph.D. thesis University of North Carolina at Chapel Hill

[8] Dubey A, Reid L B and Fisher R 2008 *Physica Scripta* **132** 014046

[9] de Supinski B R, Alam S, Bailey D H, Carrington L, Daley C, Dubey A, Gamblin T, Gunter D, Hovland P, Jagode H, Karavanic K, Marin G, Mellor-Crummey J, Moore S, Norris B, Oliker L, Olschanowsky C, Roth P C, Schulz M, Shende S, Snavely A, Spear W, Tikir M, Vetter J, Worley P and Wright N 2009 Modeling the Office of Science ten year facilities plan: The PERI Architecture Tiger Team *These proceedings*